exMath Manual

version 0.1b - written by Laurens Ruijtenberg

Table of Contents

[Introduction 3](#_Toc241607379)

[1 Basic concepts 4](#_Toc241607380)

[1.1 Delegates 4](#_Toc241607381)

[1.2 Execution 4](#_Toc241607382)

[2 Basic operation 6](#_Toc241607383)

[2.1 Sum operations 6](#_Toc241607384)

[2.2 Basic functions 7](#_Toc241607385)

[3 Deratives 8](#_Toc241607386)

# Introduction

So what is exMath? ExMath is not just a collection of classes that simply perform a series of mathematical operations and return an input.

ExMath is a framework and a collection of tools to enable the programmer to easily program complex mathematical operations with a minimum amount of code. Aside from the framework, exMath also offers tools in the form of classes that help you out with writing trivial or repetitive mathematical tasks.

Other math libraries are simply wrappers around other libraries. ExMath is designed to make optimal use of the .Net framework whenever possible. This means mathematical operations are executed using the power of delegates, the TPL framework and LINQ whenever possible and suitable. The exMath framework has been unit-tested from the ground up.

**Before you read any further**

This manual will not explain all the used mathematical concepts. I assume you are familiar with math, calculus, etc.

# 1 Basic concepts

In essence all the tools within the exMath framework rely on two essential parts:

* Delegates.
* The execution engine.

## Delegates

Delegates are used to define mathematical functions. There are three main types:

1. public delegate double MathFunction(double x);
2. public delegate double MathFunction2(double x, double y);
3. public delegate double MathFunction3(double x, double y, double z);

For now we will look at the first type. This delegate type represents a mathematical function in the form of:

Note: input and output types are always defined as doubles.

**Example C# MathFunction delegate**

public static double sqr(double x)

{

return x \* x;

}

This is a representation of the following mathematical function:

## Execution

So how do we effectively execute this function? Of course we could simply call it by using:

double result = sqr(2);

This works perfectly well if we want to execute the function using a single input variable, but for most mathematical applications we have to process large amounts of input data. In the case of this little function, the solution to this problem is pretty simple; we can write a loop to iterate through all the input values and execute the function. The results could be stored in a List for example.

But as functions get larger, so does all the plumbing code that makes the whole thing work. Before you know it you need to deal with nested loops, finding out ways to combine results or worse. So how do we deal with that problem? The exMath library offers an execution engine that makes sure that every function is executed in the most optimal way. It will also combine all the results in a single array, so you can easily iterate through all the results. Besides simply executing the function at hand, it also caches its results. This is useful when dealing with heavy computational tasks. If a function is called for a second time using input values that have been calculated before, it will look up the corresponding results in the ‘Global Function Cache’. This way, we don’t have to actually calculate the result again and a lookup operation is being done instead.

The execution engine is located in the Functions class, which contains an ExecuteFunction method:

public static double[] ExecuteFunction(double[] Values, MathFunction Function)

This method takes two input parameters: Values and a Function. Values is an array of doubles which resemble the input values within the input domain of the function. Function is a delegate that represents the actual function. We can execute our previously defined sqr function in the following way:

double[] InputValues = { 1.0, 2.0, 3.0, 4.0, 5.0 };

double[] Results = Functions.ExecuteFunction(0.0, 5.0, 1.0, sqr);

# Basic operation

Although the .net framework includes a nice System.Math class, it contains only limited functionality. ExMath includes a set of basic tools to assist you in writing cleaner math related code.

## Sum operations

One of the mostly used mathematical concepts is summing up values. The way to specify this operation in a formula, is by using the sigma notation.

**Mathematical definition of a sum operation**

**Possible C# version of above definition**

double[] x = { 1.0, 2.0, 3.0, 4.0, 5.0 };

double sum = 0.0;

for (int i = 1; i <= x.Length; i++)

{

sum += x[i] \* x[i];

}

**exMath version**

double sum = BasicOperations.Sum(1.0, 5.0, 1.0, delegate(double x){

return 0.0;

});

As you can see the exMath version is more straightforward and easier to read and understand. The Sum operation/method also makes use of the Global Function Cache; so if the specified delegate has been executed before, it will retrieve it’s answer from the cache. Let’s look at the definition of a function for calculating the *‘standard deviation of the sample’*:

**Mathematical definition for ‘standard deviation of the sample’**

**Possible C# version of the above definition**

double StdDeviation(double[] values)

{

double sum = 0.0;

foreach (double value in values)

{

sum += value;

}

double Average = sum / values.Length;

double DevidedByN = 1.0 / ((double)values.Length - 1);

double sum2 = 0.0;

foreach (double x in values)

{

double deviation = x - Average;

sum2 += deviation \* deviation;

}

return Math.Sqrt(DevidedByN \* sum2);

}

**exMath version**

public static double StandardDeviation(double[] Values)

{

double Average = Mean(Values);

double DevidedByN = 1.0 / ((double)Values.Length - 1);

double Sum = BasicOperations.Sum(Values, delegate(double x){

double deviation = x - Average;

return deviation \* deviation;

});

return Math.Sqrt(DevidedByN \* Sum);

}

## 2.2 Basic functions

Because some essential basic math functions are missing in System.Math, I’ve included some in the BasicOperations class:

* Cot
* Csc
* Csc2
* Faculty
* FlipSign
* Sec
* Sec2
* Square

# Deratives

The exMath library allows you to create derative objects based on functions. As we all know; the rate of change of a function at any point x can be calculated by using the function’s derative, where f’(x) is the derative function of f(x).

A derative from a derative is:

We can use the derative class to create a derative generator for any function, like so:

Derative d = new Derative(someFunction);

We are now able to call the derative function by using:

d.DerativeFunction(x);

Where x is some point on the x-axis somewhere in the domain of the function. We can also create a derative from another derative. You can go on and nest as many deratives as you want:

Derative d1 = new Derative(someFunction);

Derative d2 = new Derative(d1);

Derative d3 = new Derative(d2);

The Derative class also adheres to the several derative rules:

**The constant multiple rule**

Every derative object can be multiplied by some constant of type double:

will result in

C# Code:

Derative d1 = new Derative(f);

Derative d2 = d1 \* c;

**The sum rule**

Two deratives can be added together to get a new derative:

will result in

C# Code:

Derative d1 = new Derative(f);

Derative d2 = new Derative(g);

Derative d3 = d1 + d2;

**The difference rule**

Two deratives can be subtracted from each other to get a new derative:

will result in

C# Code:

Derative d1 = new Derative(f);

Derative d2 = new Derative(g);

Derative d3 = d1 – d2;

Basic statistics

The exMath library has support for the following basic statistical functions:

* Mean: Returns the average value in the input collection.
* Mode: Returns the most used value in the input collection.
* Min: Returns the lowest value in the input collection.
* Max: Returns the highest value in the input collection.
* Variance (sample-based) : Returns the variance based on the input collection.
* Variance (population-based): Returns the variance based on the input collection.
* StandardDeviation (sample-based): Returns the standard deviation based on the input collection.
* Correlation: Returns the amount of correlation based on the input collection.